Grab Documentation

Release 0.6

Gregory Petukhov

August 18, 2015

Contents

1 Table of Contents **3**

1.1 Grab User Manual 3

1.2 Grab::Spider User Manual 31

1.3 API Reference 39

2 Indices and tables **43**

Python Module Index **45**

Grab is python framework for building web scrapers. With Grab you can build web scrapers of various complexity: from simple 5-line scripts to complex asynchronous web-site crawlers processing millions of web pages. Grab provides API for performing network requests and for handling received content e.g. interacting with DOM tree of the HTML document.

There are two main parts in Grab library:

1) The single request/response API that allows you build network request, perform it and work with the received content. That API is a wrapper of the pycurl and lxml libraries.

2) The Spider API to build asynchronous web crawlers. You write class that define handlers for each type of network request. Each handler could spawn new network requests. Network requests are processed simultaneusly with a pool of asynchronous web sockets.

CHAPTER 1

Table of Contents

1.1 Grab User Manual

1.1.1 Grab Installaion

Installation on Linux

Run the command:

```
pip install -U Grab
```

This command will install Grab and all dependencies. Be aware that you need to have some libraries installed in your system to successfully build lxml and pycurl dependencies.

To build pycurl successfully you need to install:

```
apt-get install libcurl4-openssl-dev
```

To build lxml successfully you need to install:

```
apt-get install libxml2-dev libxslt-dev
```

Installation on Windows

1. Install lxml. You can get lxml here: http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml
2. Install pycurl. You can get pycurl here: http://www.lfd.uci.edu/~gohlke/pythonlibs/#pycurl
3. Install Grab

Download the Grab package from https://pypi.python.org/pypi/grab, unpack it and run:

```
python setup.py install
```

If you use Python 2.x, then you might get an error while using *python setup.py install*. There is a bug in python 2.7.6. Delete it, download python 2.7.5 and try to install Grab again.

You can also install Grab via pip. If you don't have pip installed, install pip first. Download the file get-pip.py from https://bootstrap.pypa.io/get-pip.py and then run this command:

```
python get-pip.py
```

Now you can install Grab via pip with this command:

```
python -m pip install grab
```

Installation on FreeBSD

Run the command:

```
pip install -U Grab
```

You can also install Grab from FreeBSD ports (thanks to Ruslan Makhmatkhanov):

- To install the port: cd /usr/ports/devel/py-grab/ && make install clean
- To add the package: pkg_add -r py27-grab

Installation on MacOS

Run the command:

```
pip install -U Grab
```

Dependencies

All required dependencies should be installed automatically if you install Grab with pip. Here is actual list of Grab dependencies:

```
lxml
pycurl
selection
weblib
six
```

Upgrade Grab from 0.5.x version to 0.6.x

In Grab 0.6.x some features were moved out into separate packages. If you install/upgrade Grab with pip command all dependencies should be installed automatically. Anyway, if you have some ImportError issues then try to install dependencies manually with the command:

```
pip install -U lxml pycurl selection weblib six
```

1.1.2 Testing Grab Framework

To run full set of Grab tests you should install additional dependencies listed in files requirements_dev.txt and requirements_dev_backend.txt.

To run all tests run the command:

```
./runtest.py --test-all --extra --backend-mongo --backend-mysql --backend-redis --backend-postgres
```

Controlling what parts of Grab to test

You can run tests for specific part of Grab framework. Here is available options of runtest.py command:

```
--test-grab - Grab API tests
--test-spider - Grab::Spider API tests
--test-all - shortcut to run both Grab and Grab::Spider tests
--backend-redis - enable tests of things that work with redis
--backend-mysql - enable tests of things that work with mysql
--backend-postgresql - enable tests of things that work with postgresql
--backend-mongo - enable tests of things that work with mongo
```

If you want to run specific test case then use *-t* option. Example:

```
./runtest.py -t test.grab_api
```

Testing with Tox

To run Grab tests in different python environments you can use *tox* command:

```
tox
```

By default it run full set of tests in two environemnts: python3.4 and python2.7

You can specify concrete environment with *-e* option:

```
tox -e py34
```

To run all tests except backend tests use *-nobackend* suffix:

```
tox -e py34-nobackend,py27-nobackend
```

Database configuration

To run tests on specific machine you may need to change default database connection settings. Default settings are stored in the *test_settings.py* file. You can override any default setting in the *test_settings_local.py* file.

Travis Testing

Grab project is configured to run full set of tests for each new commit placed into the project repository. You can see status of recent commit and status of all previous commits here: https://travis-ci.org/lorien/grab

Test Coverage

To see test coverage run the commands:

```
coverage erase
coverage run --source=grab ./runtest.py --test-all --extra --backend-mongo --backend-mysql --backend-
coverage report -m
```

Also you can use shortcut:

```
make coverage
```

Grab project is configured to submit coverage statistics to coveralls.io service after each test session completed on travis-ci.org service. You can see coverage history at this URL: https://coveralls.io/r/lorien/grab

1.1.3 Grab Quickstart

Before working with Grab ensure that you have a latest version of Grab. The recommended way of installing Grab to you system is using pip utility:

```
pip install -U Grab
```

Also you should manually install lxml and pycurl libraries.

Let's get started with simple examples.

Make a request

First, you need to import Grab class:

```
>>> from grab import Grab
```

Then you can build Grab instance and make simple network request:

```
>>> from grab import Grab
>>> g = Grab()
>>> resp = g.go('http://livejournal.com/')
```

Now, we have a *Response* object which provides interface to response's content, cookies, headers and other things.

We've just made a GET requests. To make reqests of other type you need to configure Grab instance via *setup* method with *method* option:

```
>>> g.setup(method='put')
>>> g.setup(method='delete')
>>> g.setup(method='options')
>>> g.setup(method='head')
```

Let's see small example of HEAD request:

```
>>> g = Grab()
>>> g.setup(method='head')
>>> resp = g.go('http://google.com/robots.txt')
>>> print len(resp.body)
0
>>> print resp.headers['Content-Length']
1776
```

Creating POST requests

When you build site scrapers or work with network APIs it is a common task to create POST requests. You can build POST request using *post* option:

```
>>> g = Grab()
>>> g.setup(post={'username': 'Root', 'password': 'asd7DD&*ssd'})
>>> g.go('http://example.com/log-in-form')
```

Another common task is to get a web form, fill it and submit it. Grab provides API to work with forms:

```
>> g = Grab()
>> g.go('http://example/com/log-in')
>> g.set_input('username', 'Foo')
>> g.set_input('password', 'Bar')
>> g.submit()
```

Then you call *submit* method Grab build POST request with values which you've passed with *set_input* methods. If you did not specify values for some form's elements then Grab use their default values.

Grab also provide interface to upload files:

```
>>> from grab import Grab, UploadFile
>>> g = Grab()
>>> g.setup(post={'name': 'Flower', 'file': UploadFile('/path/to/image.png')})
>>> g.submit()
```

Also you can upload files via form API:

```
>>> from grab import Grab, UloadFile
>>> g = Grab()
>>> g.go('http://example.com/some-form')
>>> g.set_input('name', 'A flower')
>>> g.set_input('file', UploadFile('/path/to/image.png'))
>>> g.submit()
```

Response Content

Consider the simple page retreiving code again:

```
>>> g = Grab()
>>> resp = g.go('http://google.com/')
```

To get response content as unicode use:

```
>>> resp.unicode_body()
```

Note that grab automatically detect the charset of response content and convert it to the unicode. You can get detected charset with:

```
>>> resp.charset
```

If you need original response body then use:

```
>>> resp.body
```

Oiriginal content is useful if you need to save some binary content like image::

```
>>> resp = g.go('http://example.com/some-log.png')
>>> open('logo.png', 'w').write(resp.body)
```

The *gzip* and *deflate* encodings are automatically decoded.

Response Status Code

TO BE CONTINUED

1.1.4 Request Methods

You can make request of any HTTP method you need. By default Grab makes GET request.

GET Request

GET method is default request method.

```
g = Grab()
g.go('http://example.com/')
```

If you need to pass arguments in query string, then you have to build URL manually:

```
from urllib import urlencode

g = Grab()
qs = urlencode({'foo': 'bar', 'arg': 'val'})
g.go('http://dumpz.org/?%s' % qs)
```

If your URL contains unsafe characters then you have to escape them manually.

```
from urllib import quote

g = Grab()
url = u'https://ru.wikipedia.org/wiki/'
g.go(quote(url.encode('utf-8')))
```

POST Request

To make POST request you have to specify POST data with *post* option. Usually, you will want to use dictionary value:

```
g = Grab()
g.go('http://example.com/', post={'foo': 'bar'})
```

You can pass unicode strings and numbers in values of *post* dict, they will be converted to bytes string automatically. If you want to specify charset that will be used to convert unicode to byte string, then use *request_charset* option

```
g = Grab()
g.go('http://example.com/', post={'who': u'  '},
     charset='cp1251')
```

If the *post* option is a string then it is submitted in request as-is:

```
g = Grab()
g.go('http://example.com/', post='raw data')
```

If you want to pass multiple values with same key use the list of tuples:

```
g = Grab()
g.go('http://example.com/', post=[('key', 'val1'), ('key', 'val2')])
```

By default, Grab compose POST request with 'application/x-www-form-urlencoded' encoding method. To enable *multipart/form-data* method use *post_multipart* method instead of *post*:

```
g = Grab()
g.go('http://example.com/', post_multipart=[('key', 'val1'),
                                            ('key', 'val2')])
```

To upload file use *grab.upload.UploadFile* class:

```
g = Grab()
g.go('http://example.com/',
     post_multipart={'foo': 'bar', 'file': UploadFile('/path/to/file')})
```

PUT Request

To make PUT request use both *post* and *method* options:

```
g = Grab()
g.go('http://example.com/', post='raw data', method='put')
```

Other Methods

To make DELETE, OPTIONS and other HTTP requests, use the *method* option.

```
g = Grab()
g.go('http://example.com/', method='options')
```

1.1.5 Setting up the Grab Request

To set up parameters of network request you need to build Grab object and configure it. You can do both things in same time:

```
g = Grab(url='http://example.com/', method='head', timeout=10,
         user_agent='grab')
g.request()
```

Or you can build Grab object with some initial settings and configure it later:

```
g = Grab(timeout=10, user_agent='grab')
g.setup(url='http://example.com', method='head')
g.request()
```

Also you can pass settings as parameters to *request* or *go* method:

```
g = Grab(timeout=10)
g.setup(method='head')
g.request(url='http://example.com', user_agent='grab')
# OR
g.go('http://example.com', user_agent='grab')
```

Methods *request* and *go* are almost same except one small thing. You do not need to specify the explicit name of first argument of *go* method. First argument of *go* method is always *url*. Except this thing all other named arguments of *go* and *request* are just passed to *setup* function.

Full list of available settings you can see in grab_settings

Grab Config Object

Every time you configure a Grab instance, all options go to the special object *grab.config* that holds all Grab instance settings. You can receive a copy of the config object and also you can setup a Grab instance with the config object:

```
>>> g = Grab(url='http://google.com/')
>>> g.config['url']
'http://google.com/'
>>> config = g.dump_config()
>>> g2 = Grab()
>>> g2.load_config(config)
>>> g2.config['url']
'http://google.com/'
```

The Grab config object is simply a *dict* object some of which values are also *dict* objects.

Grab Instance Cloning

If you need to copy a Grab object there is a more elegant way than using *dump_config* and *load_config* methods:

```
g2 = g1.clone()
```

Instance *g2* gets the same state as instance *g1*. In particular, *g2* gets same cookies.

There is also *adopt*, which does the opposite of the *clone* method:

```
g2.adopt(g1)
```

The *g2* instance receives the state of *g1* instance.

Setting Up the Pycurl Object

Sometimes you need more detailed control of network requests than Grab allows. In such cases you can configure the pycurl object directly. All Grab's network features are only the interface to the pycurl library. Any available Grab option just sets some option of the pycurl object. Here is a simple example of how to change the type of the HTTP authentication:

```
>>> import pycurl
>>> from grab import Grab
>>> g = Grab()
>>> g.setup(userpwd='root:123')
>>> g.transport.curl.setopt(pycurl.HTTPAUTH, pycurl.HTTPAUTH_NTLM)
```

1.1.6 Grab Options

Network options

url

Type string

Default None

The URL of the requested web page. You can use relative URLS, in which case Grab will build the absolute url by joining the relative URL with the URL or previously requested document. Be aware that Grab does not automatically escape unsafe characters in the URL. This is a design feature. You can use *urllib.quote* and *urllib.quote_plus* functions to make your URLs safe.

More info about valid URLs is in RFC 2396.

timeout

Type int

Default 15

Maximum time for a network operation. If it is exceeded, GrabNetworkTimeout is raised.

connect_timeout

Type int

Default 3

Maximum time for connection to the remote server and receipt of an initial response. If it is exceeded, GrabNetworkTimeout is raised.

follow_refresh

Type bool

Default False

Automatically follow the URL specified in <meta http-equiv="refresh"> tag.

follow_location

Type bool

Default True

Automatically follow the location in 301/302 response.

interface

Type string

Default None

The network interface through which the request should be submitted.

To specify the interface by its OS name, use "if!***" format, e.g. "if!eth0". To specify the interface by its name or ip address, use "host!***" format, e.g. "host!127.0.0.1" or "host!localhost".

See also the pycurl manual: http://curl.haxx.se/libcurl/c/curl_easy_setopt.html#CURLOPTINTERFACE

redirect_limit

Type int

Default 10

Set the maximum number of redirects that Grab will do for one request. Redirects follow the "Location" header in 301/302 network responses, and also follow the URL specified in meta refresh tags.

userpwd

Type string

Default None

The username and the password to send during HTTP authorization. The value of that options is the string of format "username:password".

HTTP Options

user_agent

Type string

Default see below

Sets the content of the "User-Agent" HTTP-header. By default, Grab randomly chooses a user agent from the list of real user agents that is built into Grab itself.

user_agent_file

Type string

Default None

Path to the text file with User-Agent strings. If this option is specified, then Grab randomly chooses one line from that file.

method

Type string

Default "GET"

Possible values "GET", "POST", "PUT", "DELETE"

The HTTP request method to use. By default, GET is used. If you specify *post* or *multipart_post* options, then Grab automatically changes the method to POST.

post

Type sequence of pairs or dict or string

Default None

Data to be sent with the POST request. Depending on the type of data, the corresponding method of handling that data is selected. The default type for POST requests is "application/x-www-form-ulencoded".

In case of *dict* or sequence of pairs, the following algorithm is applied to each value:

- objects of *grab.upload.UploadFile* class are converted into pycurl structures
- unicode strings are converted into byte strings
- None values are converted into empty strings

If *post* value is just a string, then it is placed into the network request without any modification.

multipart_post

Type sequence of pairs or dict

Default None

Data to be sent with the POST request. This option forces the POST request to be in "multipart/form-data" form.

headers

Type dict

Default None

Additional HTTP-headers. The value of this option will be added to headers that Grab generates by default. See details in *Work with HTTP Headers*.

common_headers

Type dict

Default None

By default, Grab generates some common HTTP headers to mimic the behaviour of a real web browser. If you have trouble with these deafult headers, you can specify your own headers with this option. Please note that the usual way to specify a header is to use the *headers* option. See details in *Work with HTTP Headers*.

reuse_cookies

Type bool

Default True

If this option is enabled, then all cookies in each network response are stored locally and sent back with further requests to the same server.

cookies

Type dict

Default None

Cookies to send to the server. If the option *reuse_cookies* is also enabled, then cookies from the *cookies* option will be joined with stored cookies.

cookiefile

Type string

Defaul None

Before each request, Grab will read cookies from this file and join them with stored cookies. After each response, Grab will save all cookies to that file. The data stored in the file is a dict serialized as JSON.

referer

Type string

Default see below

The content of the "Referer" HTTP-header. By default, Grab builds this header with the URL of the previously requested document.

reuse_referer

Type bool

Default True

If this option is enabled, then Grab uses the URL of the previously requested document to build the content of the "Referer" HTTP header.

Proxy Options

proxy

Type string

Default None

The address of the proxy server, in either "domain:port" or "ip:port" format.

proxy_userpwd

Type string

Default None

Security data to submit to the proxy if it requires authentication. Form of data is "username:password"

proxy_type

Type string

Default None

Type of proxy server. Available values are "http", "socks4" and "socks5".

Response Processing Options

encoding

Type string

Default "gzip"

List of methods that the remote server could use to compress the content of its response. The default value of this option is "gzip". To disable all compression, pass the empty string to this option.

document_charset

Character set of the document's content. By default, document charset is detected automatically. In case the character set is incorrectly determined, you can specify it with this option. The value you specified will be used to convert the bytes in the body of the document to the character set specified in _option_charset. After that the converted body will be passed to lxml to build the DOM tree. _option_document_charset is also used to encode non-ascii symbols in POST data.

Type string

Default None

charset

Character set that the content of the document should be converted to. This option is only useful in case *document_charset* is not 'utf-8' (e.g., if content is a *bytestring*). Otherwise, this option won't affect the resulting content.

Type string

Default 'utf-8'

nobody

Type bool

Default False

Ignore the body of the network response. When this option is enabled, the connection is abandoned at the moment when remote server transfers all response headers and begins to transfer the body of the response. You can use this option with any HTTP method.

body_maxsize

Type int

Default None

A limit on the maximum size of data that should be received from the remote server. If the limit is reached, the connection is abandoned and you can work with the data received so far.

lowercased_tree

type bool

Default False

Convert the content of the document to lowercase before passing it to the lxml library to build the DOM tree. This option does not affect the content of *response.body*, which always stores the original data.

strip_null_bytes

Type bool

Default True

Control the removal of null bytes from the body of HTML documents before they a re passed to lxml to build a DOM tree. lxml stops processing HTML documents at the first place where it finds a null byte. To avoid such issues Grab, removes null bytes from the document body by default. This option does not affect the content of *response.body* that always stores the original data.

body_inmemory

Type bool

Default True

Control the way the network response is received. By default, Grab downloads data into memory. To handle large files, you can set *body_inmemory=False* to download the network response directly to the disk.

storage_dir

Type bool

Default None

If you use *body_inmemory=False*, then you have to specify the directory where Grab will save network requests.

body_storage_filename

Type string

Default None

If you use *body_inmemory=False*, you can let Grab automatically choose names for the files where it saves network responses. By default, Grab randomly builds unique names for files. With the *body_storage_filename* option, you can choose the exact file name to save response to. Note that Grab will save every response to that file, so you need to change the *body_storage_filename* option before each new request, or set it to None to enable default randomly generated file names.

content_type

Type string

Default "html"

Available values "html" and "xml"

This option controls which lxml parser is used to process the body of the response. By default, the html parser is used. If you want to parse XML, then you may need to change this option to "xml" to force the use of an XML parser which does not strip the content of CDATA nodes.

fix_special_entities

Type bool

Default True

Fix &#X; entities, where X between 128 and 160. Such entities are parsed by modern browsers as windows-1251 entities, independently of the real charset of the document. If this option is True, then such entities will be replaced with appropriate unicode entities, e.g.: — -> —

Debugging

log_file

Type string

Default None

Path to the file where the body of the recent network response will be saved. See details at *Saving the content of requests and responses*.

log_dir

Type string

Default None

Directory to save the content of each response in. Each response will be saved to a unique file. See details at *Saving the content of requests and responses*.

verbose_logging

Type bool

Default False

This option enables printing to console of all detailed debug info about each pycurl action. Sometimes this can be useful.

debug_post

Type bool

Default False

Enable logging of POST request content.

1.1.7 Debugging

Using the logging module

The easiest way to see what is going on is to enable DEBUG logging messages. Write the following code at every entry point to your program:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
```

That logging configuration will output all logging messages to console, not just from Grab but from other modules too. If you are interested only in Grab's messages:

```
>>> import logging
>>> logger = logging.getLogger('grab')
>>> logger.addHandler(logging.StreamHandler())
>>> logger.setLevel(logging.DEBUG)
```

You can also use a *default_logging* function that configures logging as follows:

- all messages of any level except from Grab modules are printed to console
- all "grab*" messages with level INFO or higher are printed to console
- all "grab*" messages of any level are saved to /tmp/grab.log
- all "grab.network*" messages (usually these are URLs being requested) of any level are saved to /tmp/grab.network.log

Usage of *default_logging* function is simple:

```
>>> from weblib.logs import default_logging
>>> default_logging()
```

Logging messages about network request

For each network request, Grab generates the "grab.network" logging message with level DEBUG. Let's look at an example:

```
[5864] GET http://www.kino-govno.com/movies/rusichi via 188.120.244.68:8080 proxy of type http with a
```

We can see the requested URL and also that request has ID 5864, that the HTTP method is GET, and that the request goes through a proxy with authorization. For each network request Grab uses the next ID value from the sequence that is shared by all Grab instances. That does mean that even different Grab instances will generates network logging messages with unique ID.

You can also turn on logging of POST request content. Use the *debug_post* option:

```
>>> g.setup(debug_post=True)
```

The output will be like this:

```
[01] POST http://yandex.ru
POST request:
foo                       : bar
name                      : Ivan
```

Saving the content of requests and responses

You can ask Grab to save the content of each network response to the file located at the path passed as the *log_file* option:

```
>>> g.setup(log_file='log.html')
```

Of course, each new resposne will overwrite the content of the previous response.

If you want to log all traffic, then consider using the *log_dir* option, which tells Grab to save the contents of all responses to files inside the specified directory. Note that each such file will contain a request ID in its filename. For each response, there will be two files: XXX.log and XXX.html. The file XXX.html contains the raw response. Even if you requested an image or large movie, you'll get its raw content in that file. The file XXX.log contains headers of network response. If you configure Grab with *debug=True*, the file XXX.log will also contain request headers.

1.1.8 Work with HTTP Headers

Custom headers

If you need to submit custom HTTP headers, you can specify any number of them via *headers* option. A common case is to emulate an AJAX request:

```
>>> g = Grab()
>>> g.setup(header={'X-Requested-With': 'XMLHttpRequest'})
```

Bear in mind, that except headers in *headers* option (that is empty by default) Grab also generates a bunch of headers to emulate a typical web browser. At the moment of writing these docs these headers are:

- Accept
- Accept-Language
- Accept-Charset
- Keep-Alive
- Except

If you need to change one of these headers, you can override its value with the *headers* option. You can also subclass the Grab class and define your own *common_headers* method to completely override the logic of generating these extra headers.

User-Agent header

By default, for each request Grab randomly chooses one user agent from a builtin list of real user agents. You can specify the exact User-Agent value with the *user_agent* option. If you need to randomly choose user agents from your own list of user agents, then you can put your list into a text file and pass its location as *user_agent_file*.

Referer header

To specify the content of the Referer header, use the *referer* option. By default, Grab use the URL of previously request document as value of Referer header. If you do not like this behaviour, you can turn it off with *reuse_referer* option.

HTTP Authentication

To send HTTP authentication headers, use the *userpwd* option with a value of the form "username:password".

1.1.9 Processing the Response Body

In this document, options related to processing the body of network response are discussed.

Partial Response Body Download

If you do not need the response body at all, use *nobody* option. When it is enabled, Grab closes the network connection to the server right after it receives all response headers from the server. This is not the same as sending a GET request. You can submit a request of any type, e.g. POST, and not download the response body if you do not need it.

Another option to limit body processing is *body_maxsize*. It allows you to download as many bytes of the response body as you need, and then closes the connection.

Note that neither of these options break the processing of the response into a Python object. In both cases you get a response object with a body attribute that contains only part of the response body data - whatever was received before connection interrupted.

Response Compression Method

You can control the compression of the server response body with *encoding*. The default value is "gzip". That means that Grab sends "Accept-Encoding: gzip" to the server, and if the server answers with a response body packed with gzip then Grab automatically unpacks the gzipped body, and you have unpacked data in the *response.body*. If you do not want the server to send you gziped data, use an empty string as the value of *encoding*.

1.1.10 File Uploading

To upload file you should use *UploadFile* or *UploadContent* classes.

UploadFile example:

```
from grab import Grab, UploadFile

g = Grab()
g.setup(post={'image': UploadFile('/path/to/image.jpg')})
g.go('http://example.com/form.php')
```

UploadContent example:

```
from grab import Grab, UploadContent

g = Grab()
g.setup(post={'image': UploadContent('......', filename='image.jpg')})
g.go('http://example.com/form.php')
```

Form Processing

You can use *UploadFile* and *UploadContent* in all methods that set values in form fields:

```
from grab import Grab, UploadContent

g = Grab()
g.go('http://example.com/form.php')
g.doc.set_input('image', UploadFile('/path/to/image.jpg'))
g.doc.submit()
```

Custom File Name

With both *UploadFile* and *UploadContent* you can use custom filename.

If you do not specify filename then:

- *UploadFile* will use the filename extracted from the path to the file passed in first argument.
- *UploadContent* will generate random file name

```
>>> from grab import UploadFile, UploadContent
>>> UploadFile('/path/to/image.jpg').filename
'image.jpg'
>>> UploadFile('/path/to/image.jpg', filename='avatar.jpg').filename
'avatar.jpg'
>>> UploadContent('.....').filename
'528e418951'
>>> UploadContent('.....', filename='avatar.jpg').filename
'avatar.jpg'
```

Custom Content Type

With both *UploadFile* and *UploadContent* you can use custom content type.

If you do not specify content type then filename will be used to guess the content type e.g. "image.jpg" will have "image/jpeg" conent type and "asdfasdf" will be just a "application/octet-stream"

```
>>> from grab import UploadFile, UploadContent
>>> UploadFile('/path/to/image.jpg').content_type
'image/jpeg'
>>> UploadFile('/path/to/image.jpg', content_type='text/plain').content_type
'text/plain'
>>> UploadContent('/path/to/image.jpg').content_type
'image/jpeg'
>>> UploadContent('...', content_type='text/plain').content_type
'text/plain'
```

1.1.11 Redirect Handling

Grab supports two types of redirects:

- HTTP redirects with HTTP 301 and 302 status codes
- HTML redirects with the <meta> HTML tag

HTTP 301/302 Redirect

By default, Grab follows any 301 or 302 redirect. You can control the maximum number of redirects per network query with the *redirect_limit* option. To completely disable handling of HTTP redirects, set *follow_location* to False.

Let's see how it works:

```
>>> g = Grab()
>>> g.setup(follow_location=False)
>>> g.go('http://google.com')
<grab.response.Response object at 0x1246ae0>
>>> g.response.code
301
>>> g.response.headers['Location']
'http://www.google.com/'
>>> g.setup(follow_location=True)
>>> g.go('http://google.com')
<grab.response.Response object at 0x1246ae0>
>>> g.response.code
200
```

```
>>> g.response.url
'http://www.google.ru/?gws_rd=cr&ei=BspFUtS8EOWq4ATAooGADA'
```

Meta Refresh Redirect

An HTML Page could contain special tags that instructs the browser to go to a specified URL:

```
<meta http-equiv="Refresh" content="0; url=http://some/url" />
```

By default, Grab ignores such instructions. If you want automatically follow meta refresh tags, then set *follow_refresh* to True.

Original and Destination URLs

You can always get information about what URL you've requested intially and what URL you ended up with:

```
>>> g = Grab()
>>> g.go('http://google.com')
<grab.response.Response object at 0x20fcae0>
>>> g.config['url']
'http://google.com'
>>> g.response.url
'http://www.google.ru/?gws_rd=cr&ei=8spFUo32Huem4gT6ooDwAg'
```

The initial URL is stored on the config object. The destination URL is writed into *response* object.

You can even track redirect history with *response.head*:

```
>>> print g.response.head
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Fri, 27 Sep 2013 18:19:13 GMT
Expires: Sun, 27 Oct 2013 18:19:13 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

HTTP/1.1 302 Found
Location: http://www.google.ru/?gws_rd=cr&ei=IsxFUp-8CsT64QTZooDwBA
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Date: Fri, 27 Sep 2013 18:19:14 GMT
Server: gws
Content-Length: 258
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

HTTP/1.1 200 OK
Date: Fri, 27 Sep 2013 18:19:14 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
```

```
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Transfer-Encoding: chunked
```

1.1.12 Form Processing

Grab can help you process web forms. It can automatically fill all input fields that have default values, letting you fill only fields you need. The typical workflow is:

```
* request a page
* fill input fields with `set_input` method
* submit the form with `submit` method
```

When you are using *set_input* you just specify the name of an input and the value, and Grab automatically finds the form field containing the input with that name. When you call *submit*, the automatically-chosen form is submitted (the form that has the largest number of fields). You can also explicitly choose the form with the *choose_form* method.

Let's look at a simple example of how to use these form features:

```
>>> g = Grab()
>>> g.go('http://ya.ru/')
>>> g.set_input('text', 'grab lib')
>>> g.submit()
>>> g.doc.select('//a[@class="b-serp-item__title-link"]/@href').text()
'http://grablib.org/'
```

The form that has been chosen automatically is available in the *grab.form* attribute.

To specify input values you can use *set_input*, *set_input_by_id* and *set_input_by_xpath* methods.

1.1.13 Network Errors Handling

Network Errors

If a network request fails, Grab raises *grab.error.GrabNetworkError*. There are two situations when a network error exception will raise:

- the server broke connection or the connection timed out
- the response had any HTTP status code that is not 2XX or 404

Note particularly that 404 is a valid status code, and does not cause an exception to be raised.

Network Timeout

You can configure timeouts with the following options:

- connect to server timeout with *connect_timeout* option
- whole request/response operation timeout with *timeout* option

In case of a timeout, Grab raises *grab.error.GrabTimeoutError*.

1.1.14 HTML Document Charset

Why does charset matter?

By default, Grab automatically detects the charset of the body of the HTML document. It uses this detected charset to

- build a DOM tree
- convert the bytes from the body of the document into a unicode stream
- search for some unicode string in the body of the document
- convert unicode into bytes data, then some unicode data needs to be sent to the server from which the response was received.

The original content of the network response is always accessable at *response.body* attribute. A unicode representation of the document body can be obtained by calling *response.unicode_body()*:

```
>>> g.go('http://mail.ru/')
<grab.response.Response object at 0x7f7d38af8940>
>>> type(g.response.body)
<type 'str'>
>>> type(g.response.unicode_body())
<type 'unicode'>
>>> g.response.charset
'utf-8'
```

Charset Detection Algorithm

Grab checks multiple sources to find out the real charset of the document's body. The order of sources (from most important to less):

- HTML meta tag:

```
    <meta name="http-equiv" content="text/html; charset=cp1251" >
```

- XML declaration (in case of XML document):

```
    <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

- Content-Type HTTP header:

```
    Content-Type: text/html; charset=koi8-r
```

If no source indicates the charset, or if the found charset has an invalid value, then grab falls back to a default of UTF-8.

Setting the charset manually

You can bypass automatic charset detection and specify it manually with *charset* option.

1.1.15 Cookie Support

By default, Grab automatically handles all cookies it receives from the remote server. Grab remembers all cookies and sends them back in future requests. That allows you to easily implement scripts that log in to some site and then do some activity in a member-only area. If you do not want Grab to automatically process cookies, use *reuse_cookies* option to disable it.

Custom Cookies

To send some custom cookies, use the *cookies* option. The value of that option should be a dict. When you specify some cookies with *cookies* option and then fire network request, all specified cookies are bound to the hostname of the request. If you want more granular control on custom cookies, you can use the *grab.cookies* cookie manager to specify a cookie with any attributes you want:

```
>>> g = Grab()
>>> g.cookies.set(name='foo', value='bar', domain='yandex.ru', path='/host')
```

Loading/dumping cookies

To dump current cookies to a file, use *grab.cookie.CookieManager.save_to_file()*.

To load cookies from a file, use *grab.cookie.CookieManager.load_from_file()*.

Permanent file to load/store cookies

With the *cookiefile* option, you can specify the path to the file that Grab will use to store/load cookies for each request. Grab will load any cookies from that file before each network request, and after a response is received Grab will save all cookies to that file.

More details about *grab.cookies* you can get in api_grab_cookie

1.1.16 Proxy Server Support

Basic Usage

To make Grab send requests through a proxy server, use the *proxy* option:

```
g.setup(proxy='example.com:8080')
```

If the proxy server requires authentication, use the option_proxy_usrpwd option to specify the username and password:

```
g.setup(proxy='example.com:8080', proxy_userpwd='root:777')
```

You can also specify the type of proxy server: “http”, “socks4” or “socks5”. By default, Grab assumes that proxy is of type “http”:

```
g.setup(proxy='example.com:8080', proxy_userpwd='root:777', proxy_type='socks5')
```

You can always see which proxy is used at the moment in *g.config['proxy']*:

```
>>> g = Grab()
>>> g.setup(proxy='example.com:8080')
>>> g.config['proxy']
'example.com:8080'
```

Proxy List Support

Grab supports working with a list of multiple proxies. Use the *g.proxylist* attribute to get access to the proxy manager. By default, the proxy manager is created and initialized with an empty proxy list:

```
>>> g = Grab()
>>> g.proxylist
<grab.proxy.ProxyList object at 0x2e15b10>
>>> g.proxylist.proxy_list
[]
```

Proxy List Source

You need to setup the proxy list manager with details of the source that manager will load proxies from. Using the *g.proxylist.set_source* method, the first positional argument defines the type of source. Currently, two types are supported: "file" and "remote".

Example of loading proxies from local file:

```
>>> g = Grab()
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
<grab.proxy.ProxyList object at 0x2e15b10>
>>> g.proxylist.proxy_list
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
>>> g.proxylist.get_next()
>>> g.proxylist.get_next_proxy()
<grab.proxy.Proxy object at 0x2d7c610>
>>> g.proxylist.get_next_proxy().server
'example.com'
>>> g.proxylist.get_next_proxy().address
'example.com:8080'
>>> len(g.proxylist.proxy_list)
1000
```

And here is how to load proxies from the web:

```
>>> g = Grab()
>>> g.proxylist.set_source('remote', url='http://example.com/proxy.txt')
```

Automatic Proxy Rotation

By default, if you set up any non-empty proxy source, Grab starts rotating through proxies from the proxy list for each request. You can disable proxy rotation with option_proxy_auto_change option set to False:

```
>>> from grab import Grab
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> g = Grab()
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
>>> g.go('http://yandex.ru/')
DEBUG:grab.network:[02] GET http://yandex.ru/ via 91.210.101.31:8080 proxy of type http with authori
<grab.response.Response object at 0x109d9f0>
>>> g.go('http://rambler.ru/')
DEBUG:grab.network:[03] GET http://rambler.ru/ via 194.29.185.38:8080 proxy of type http with author
<grab.response.Response object at 0x109d9f0>
```

Now let's see how Grab works when *proxy_auto_change* is False:

```
>>> from grab import Grab
>>> import logging
>>> g = Grab()
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
```

```
>>> g.setup(proxy_auto_change=False)
>>> g.go('http://ya.ru')
DEBUG:grab.network:[04] GET http://ya.ru
<grab.response.Response object at 0x109de50>
>>> g.change_proxy()
>>> g.go('http://ya.ru')
DEBUG:grab.network:[05] GET http://ya.ru via 62.122.73.30:8080 proxy of type http with authorization
<grab.response.Response object at 0x109d9f0>
>>> g.go('http://ya.ru')
DEBUG:grab.network:[06] GET http://ya.ru via 62.122.73.30:8080 proxy of type http with authorization
<grab.response.Response object at 0x109d9f0>
```

Getting Proxy From Proxy List

Each time you call *g.proxylist.get_next_proxy*, you get the next proxy from the proxy list. When you receive the last proxy in the list, you'll continue receiving proxies from the beginning of the list. You can also use *g.proxylist.get_random_proxy* to pick a random proxy from the proxy list.

Automatic Proxy List Reloading

Grab automatically rereads the proxy source each *g.proxylist.reload_time* seconds. You can set the value of this option as follows:

```
>>> g = Grab()
>>> g.proxylist.setup(reload_time=3600) # reload proxy list one time per hour
```

Proxy Accumulating

Be default, Grab overwrites the proxy list each time it reloads the proxy source. You can change that behaviour:

```
>>> g.proxylist.setup(accumulate_updates=True)
```

That will setup Grab to append new proxies to existing ones.

1.1.17 Searchng the response body

String search

With the *doc.text_search* method, you can find out if the response body contains a certain string or not:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text_search(u'tes')
True
```

If you prefer to raise an exception if string was not found, then use the *doc.text_assert* method:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text_assert(u'tez')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lorien/web/grab/grab/document.py", line 109, in text_assert
    raise DataNotFound(u'Substring not found: %s' % anchor)
grab.error.DataNotFound: Substring not found: tez
```

By default, all text search methods operate with unicode; i.e., you should pass unicode arguments to these methods and these methods will search inside document's body converted to unicode. There is an option to work with raw bytes, just pass *byte=True* to any method:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text_search(b'tez', byte=True)
```

Regexp Search

You can search for a regular expression with *doc.rex_search* method that accepts compiled regexp object or just a text of regular expression:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex_search('<.+?>').group(0)
u'<h1>'
```

Method *doc.rex_text* returns you text contents of *.group(1)* of the found match object:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex_text('<.+?>(.+)<')
u'test'
```

Method *doc.rex_assert* raises *DataNotFound* exception if no match is found:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex_assert('\w{10}')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lorien/web/grab/grab/document.py", line 189, in rex_assert
    self.rex_search(rex, byte=byte)
  File "/home/lorien/web/grab/grab/document.py", line 180, in rex_search
    raise DataNotFound('Could not find regexp: %s' % regexp)
grab.error.DataNotFound: Could not find regexp: <_sre.SRE_Pattern object at 0x7fa40e97d1f8>
```

1.1.18 Pycurl Tips

Asynchronous DNS Resolving

Pycurl allows you to drive network requests asynchronously with the multicurl interface. Unfortunately, by default multicurl do not handle DNS requests asynchronously. Than means that every DNS request blocks other network activity. You can manage it by building curl from source and configuring it to use the ares library, which knows how to do asynchronous DNS requests.

First, you need to download curl sources from http://curl.haxx.se/download.html. Then unpack source code and run the command:

```
$ ./configure --prefix=/opt/curl --enable-ares
```

We use a custom prefix because we do not want to mix up our custom curl with the curl lib that could be already installed in your system. Do not forget to install cares packages before configuring curl with ares:

```
$ apt-get install libc-ares-dev
```

If ./configure command has finished successfully, then run:

```
$ make
$ make install
```

Now you have customized the curl library files at /opt/curl/lib.

To let your python script know that you want to use this custom curl lib, use the following feature:

```
$ LD_LIBRARY_PATH="/opt/curl/lib" python your_script.py
```

You can manually check that you used a curl compiled with ares support:

```
>>> import pycurl
>>> pycurl.version
'libcurl/7.32.0 OpenSSL/1.0.1e zlib/1.2.7 c-ares/1.10.0 libidn/1.25'
```

You should see something like "c-ares/1.10.0" if everything was correct.

Supported Protocols

By default, pycurl supports a ton of protocols including SMTP, POP3, SSH, media streams, and FTP. If you do not need all this crap, you can disable it at the configure stage. Here is example of what you can do:

```
./configure --without-libssh2 --disable-ipv6 --disable-ldap --disable-ldaps\
            --without-librtmp --disable-rtsp --disable-ftp --disable-dict\
            --disable-telnet --disable-tftp --disable-pop3 --disable-imap\
            --disable-smtp --disable-gopher --without-winssl --without-darwinssl\
            --without-winidn
```

To see all available options, just run the command:

```
./configure --help
```

Multicurl and SOCKS proxy

This combination just does not work. Use HTTP proxies with multicurl.

1.1.19 Work With Network Response

Response Object

The result of doing a network request via Grab is a *Response* object.

You get a Response object as a result of calling to *g.go*, *g.request* and *g.submit* methods. You can also access the response object of a recent network query via the *g.response* attribute:

```
>>> from grab import Grab
>>> g = Grab()
>>> g.go('http://google.com')
<grab.response.Response object at 0x2cff9f0>
>>> g.response
<grab.response.Response object at 0x2cff9f0>
```

You can find a full list of response attributes in the Response API document. Here are the most important things you should know:

body original body contents of HTTP response

code HTTP status of response

headers HTTP headers of response

charset charset of the response

cookies cookies in the response

url the URL of the response document. In case of some automatically processed redirect, the *url* attribute contains the final URL.

name_lookup_time time spent to resolve host name

connect_time time spent to connect to remote server

total_time total time spent to complete the request

download_size size of received data

upload_size size of uploaded data except the HTTP headers

Now, a real example:

```
>>> from grab import Grab
>>> g = Grab()
>>> g.go('http://wikipedia.org')
<grab.response.Response object at 0x1ff99f0>
>>> g.response.body[:100]
'<!DOCTYPE html>\n<html lang="mul" dir="ltr">\n<head>\n<!-- Sysops: Please do not edit the main templ
>>> g.response.code
200
>>> g.response.headers['Content-Type']
'text/html; charset=utf-8'
>>> g.response.charset
'utf-8'
>>> g.response.cookies
<grab.cookie.CookieManager object at 0x1f6b248>
>>> g.response.url
'http://www.wikipedia.org/'
>>> g.response.name_lookup_time
0.103931
>>> g.response.connect_time
0.221996
>>> g.response.total_time
0.7791399999999999
>>> g.response.download_size
11100.0
>>> g.response.upload_size
0.0
```

Now let's see some useful methods available in the response object:

unicode_body() this method returns the response body converted to unicode

copy() returns a clone of the response object

save(path) saves the response object to the given location

json treats the response content as json-serialized data and de-serializes it into a python object. Actually, this is not a method, it is a property.

url_details() return the result of calling *urlparse.urlsplit* with *response.url* as an argument.

query_param(name) extracts the value of the *key* argument from the query string of *response.url*.

1.2 Grab::Spider User Manual

Grab::Spider is a framework to build well-structured asyncronous web-site crawlers.

1.2.1 What is Grab::Spider?

The Spider is a framework that allow to describe web-site crawler as set of handlers. Each handler handles only one specific type of web pages crawled on web-site e.g. home page, user profile page, search results page. Each handler could spawn new requests which will be processed in turn by other handlers.

The Spider process network requests asynchronously. There is only one process that handles all network, business logic and HTML-processing tasks. Network requests are performed by multicurl library. In short, when you create new network request it is processed by multicurl and when the response is ready, then the corresponding handler from your spider classs is called with result of networ request.

Each handler receives two arguments. First argument is a Grab object, that contains all data bout network request and response. The second argument is Task object. Whenever you need to send network request you create Task object.

Let's check out simple example. Let's say we want to go to habrahabr.ru web-site, read titles of recent news, then for each title find the image on images.yandex.ru and save found data to the file.

```
# coding: utf-8
import urllib
import csv
import logging

from grab.spider import Spider, Task

class ExampleSpider(Spider):
    # List of initial tasks
    # For each URL in this list the Task object will be created
    initial_urls = ['http://habrahabr.ru/']

    def prepare(self):
        # Prepare the file handler to save results.
        # The method `prepare` is called one time before the
        # spider has started working
        self.result_file = csv.writer(open('result.txt', 'w'))

        # This counter will be used to enumerate found images
        # to simplify image file naming
        self.result_counter = 0

    def task_initial(self, grab, task):
        print 'Habrahabr home page'

        # This handler for the task named `initial i.e.
        # for tasks that have been created from the
        # `self.initial_urls` list

        # As you see, inside handler you can work with Grab
        # in usual way i.e. just if you have done network request
        # manually
        for elem in grab.doc.select('//h1[@class="title"]'
                                    '/a[@class="post_title"]'):
            # For each title link create new Task
            # with name "habrapost"
```

```
            # Pay attention, that we create new tasks
            # with yield call. Also you can use `add_task` method:
            # self.add_task(Task('habrapost', url=...))
            yield Task('habrapost', url=elem.get('href'))

    def task_habrapost(self, grab, task):
        print 'Habrahabr topic: %s' % task.url

        # This handler receives results of tasks we
        # created for each topic title found on home page

        # First, save URL and title into dictionary
        post = {
            'url': task.url,
            'title': grab.xpath_text('//h1/span[@class="post_title"]'),
        }

        # Next, create new network request to search engine to find
        # the image related to the title.
        # We pass info about the found publication in the arguments to
        # the Task object. That allows us to pass information to next
        # handler that will be called for found image.
        query = urllib.quote_plus(post['title'].encode('utf-8'))
        search_url = 'http://images.yandex.ru/yandsearch'\
                     '?text=%s&rpt=image' % query
        yield Task('image_search', url=search_url, post=post)

    def task_image_search(self, grab, task):
        print 'Images search result for %s' % task.post['title']

        # In this handler we have received result of image search.
        # That is not image! This is just a list of found images.
        # Now, we take URL of first image and spawn new network
        # request to download the image.
        # Also we pass the info about pulication, we need it be
        # available in next handler.
        image_url = grab.xpath_text('//div[@class="b-image"]/a/img/@src')
        yield Task('image', url=image_url, post=task.post)

    def task_image(self, grab, task):
        print 'Image downloaded for %s' % task.post['title']

        # OK, this is last handler in our spider.
        # We have received the content of image,
        # we need to save it.
        path = 'images/%s.jpg' % self.result_counter
        grab.response.save(path)
        self.result_file.writerow([
            task.post['url'].encode('utf-8'),
            task.post['title'].encode('utf-8'),
            path
        ])
        # Increment image counter
        self.result_counter += 1


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
```

```
    # Let's start spider with two network concurrent streams
    bot = ExampleSpider(thread_number=2)
    bot.run()
```

In this example, we have considered the simple spider. I hope you have got idea about how it works. See other parts of *Grab::Spider User Manual* to get detailed description of spider features.

1.2.2 Task Object

Any Grab::Spider crawler is a set of handlers that process network responses. Each handler can spawn new network requests or just process/save data. The spider add each new request to task queue and process this task when there is free network stream. Each task is assigned a name that defines its type. Each type of task are handles by specific handler. To find the handler the Spider takes name of the task and then looks for *task_<name>* method.

For example, to handle result of task named "contact_page" we need to define "task_contact_page" method:

```
    ...
    self.add_task(Task('contact_page', url='http://domain.com/contact.html'))
    ...

def task_contact_page(self, grab, task):
    ...
```

Constructor of Task Class

Constructor of Task Class accepts multiple arguments. At least you have to specify URL of requested document OR the configured Grab object. Next, you see examples of different task creation. All three examples do the same:

```
# Using `url` argument
t = Task('wikipedia', url 'http://wikipedia.org/')

# Using Grab intance
g = Grab()
g.setup(url='http://wikipedia.org/')
t = Task('wikipedia', grab=g)

# Using configured state of Grab instance
g = Grab()
g.setup(url='http://wikipedia.org/')
config = g.dump_config()
t = Task('wikipedia', grab_config=config)
```

Also you can specify these arguments:

priority , , , .

disable_cache ,

refresh_cache ,

valid_status . 2xx, 404.

use_proxylist . .

Task Object as Data Storage

If you pass the argument that is unknown then it will be saved in the Task object. That allows you to pass data between network request/response.

There is *get* method that return value of task attribute or *None* if that attribute have not been defined.

```
t = Task('bing', url='http://bing.com/', disable_cache=True, foo='bar')
t.foo # == "bar"
t.get('foo') # == "bar"
t.get('asdf') # == None
t.get('asdf', 'qwerty') # == "qwerty"
```

Cloning Task Object

Sometimes it is usefule to create copy of Task object. For example:

```
# task.clone()
# TODO: example
```

Setting Up Initial Tasks

When you call *run* method of your spider it starts working from initial tasks. There are few ways to setup initial tasks.

initial_urls

You can specify list of URLs in *self.initial_urls*. For each URl in this list the spider will create Task object with name "initial":

```
class ExampleSpider(Spider):
    initial_urls = ['http://google.com/', 'http://yahoo.com/']
```

task_generator

More flexible way to define initial tasks is to use *task_generator* method. Its interface is simple, you just have to yield new Task objects.

There is common use case when you need to process big number of URLs from the file. With *task_generator* you can iterate over lines of the file and yield new tasks. That will save memory used by the script because you will not read whole file into the memory. Spider consumes only portion of tasks from *task_generator*. When there are free networks resources the spiders consumes next portion of task. And so on.

Example:

```
class ExampleSpider(Spider):
    def task_generator(self):
        for line in open('var/urls.txt'):
            yield Task('download', url=line.strip())
```

Explicit Ways to Add New Task

Adding Tasks With add_task method

You can use *add_task* method anywhere, even before the spider have started working:

```
bot = ExampleSpider()
bot.setup_queue()
bot.add_task('google', url='http://google.com')
bot.run()
```

Yield New Tasks

You can use yield statement to add new tasks in two places. First, in *task_generator*. Second, in any handler. Using yield is completely equal to using *add_task* method. The yielding is just a bit more beautiful:

```
class ExampleSpider(Spider):
    initial_urls = ['http://google.com']

    def task_initial(self, grab, task):
        # Google page was fetched
        # Now let's download yahoo page
        yield Task('yahoo', url='yahoo.com')

    def task_yahoo(self, grab, task):
        pass
```

Default Grab Instance

You can control the default config of Grab instances used in spider tasks. Define the *create_grab_instance* method in your spider class:

```
class TestSpider(Spider):
    def create_grab_instance(self, **kwargs):
        g = super(TestSpider, self).create_grab_instance(**kwargs)
        g.setup(timeout=20)
        return g
```

Be aware, that this method allows you to control only those Grab instances that were created automatically. If you create task with explicit grab instance it will not be affected by *create_grab_instance_method*:

```
class TestSpider(Spider):
    def create_grab_instance(self, **kwargs):
        g = Grab(**kwargs)
        g.setup(timeout=20)
        return g

    def task_generator(self):
        g = Grab(url='http://example.com')
        yield Task('page', grab=g)
        # The grab instance in the yielded task
        # will not be affected by `create_grab_instance` method.
```

Updating Any Grab Instance

With method *update_grab_instance* you can update any Grab instance, even those instances that you have passed explicitly to the Task object. Be aware, that any option configured in this method overwrites the previously configured option.

```
class TestSpider(Spider):
    def update_grab_instance(self, grab):
        grab.setup(timeout=20)

    def task_generator(self):
        g = Grab(url='http://example.com', timeout=5)
        yield Task('page', grab=g)
        # The effective timeout setting will be equal to 20!
```

1.2.3 Task Queue

Task Priorities

All new tasks are places into task queue. The Spider get tasks from task queue when there are free network streams. Each task has priority. Lower number means higher priority. Task are processed in the order of their priorities: from highest to lowest. If you do not specify the priority for the new task then it is assigned automatically. There are two algorithms of assigning default task priorities:

random random priorities

const same priority for all tasks

By default random priorities are used. You can control the algorithm of default priorities with *priority_mode* argument:

```
bot = SomeSpider(priority_mode='const')
```

Tasks Queue Backends

You can choose the storage for the task queue. By default, Spider uses python *PriorityQueue* as storage. In other words, the storage is memory. You can also used redis and mongo backends.

In-memory backend:

```
bot = SomeSpider()
bot.setup_queue() #
# OR (that is the same)
bot.setup_queue(backend='memory')
```

MongoDB backend:

```
bot = SomeSpider()
bot.setup_queue(backend='mongo', database='database-name')
```

All arguments except *backend* go to MongoDB connection constructor. You can setup database name, host name, port, authorization arguments and other things.

Redis backend:

```
bot = SomeSpider()
bot.setup_queue(backend='redis', db=1, port=7777)
```

1.2.4 Spider Cache

There is cache built in the spider. It could be helpful on development stage. When you need to scrape same documents for many times to check the results and to fix bugs. Also you can crawl whole web-site, put it into cache and then work only with cache.

Keep in mind that if the web-site is large, millions of web pages then working with cache could be slower than working with live web-site. This is because of limited disk I/O where the cache storage is hosted.

Also keep in mind the the spider cache is very simple:

- it allows to cache only GET requests
- **it does not allow to diffirentiate documents with same URL but** different cookies/headers
- it does not support max-age and other cache headers

Spider Cache Backends

You can choose what storage to use for the cache. You can use mongodb, mysql and postgresql.

MongoDB example:

```
bot = ExampleSpider()
bot.setup_cache(backend='mongo', database='some-database')
bot.run()
```

In this example the spider is configured to use mongodb as cache storage. The name of database is "some-database". The name of collection would be "cache".

All arguments except *backend*, *database* and *use_compression* go to database connection constructor. You can setup database name, host name, port, authorization arguments and other things.

Example of custom host name and port for mongodb connection:

```
.. code:: python
```

bot = SomeSpider() bot.setup_cache(backend='mongo', port=7777, host='mongo.localhost')

Cache Compression

By defalt cache compression is enabled. That means that all documents placed in the cache are compressed with gzip libary. Compression decreases the disk space required to store the cache and increases the CPU load (a bit).

1.2.5 Spider Error Handling

Rules of Network Request Handling

- If request is completed succesfully then the corresponding handler is called
- If request is failed due the network error, then the task is submitted back to the task queue
- If the request is completed and the handler is called and failed due to any error inside the handler then the task processing is aborted. This type of errors is not fatal. The handler error is logged and other requests and handlers are processed in usual way.

Network Errors

Network error is:

- error occured in process of data transmition to or back from the server e.g. connection aborted, connection timeout, server does not accep connection and so on
- data transmition has been completed but the HTTP sttus of received document differs from 2XX or from 404

Yes, by default documents with 404 status code counts as valid! That makes sense to me :) If that is not you want then you can configure custom rule to mark status as valid or failed. You have two ways.

First way is to use *valid_status* argument in Task constructor. With this argument you can only extend the default valid status. This arguments accepts list of additional valid status codes:

```
t = Task('example', url='http://example.com', valid_status=(500, 501, 502))
```

Second way is to redefine *valid_response_code* method. In this way you can implement any logic you want. Method accepts two arguments: status code and task object. Method returns boolean value, *True* means that the status code is valid:

```
class SomeSpider(Spider):
    def valid_response_code(self, code, task):
        return code in (200, 301, 302)
```

Handling of Failed Tasks

The task failed due to the network error is put back to tas queue. The number of tries is limited to the *Spider.network_try_limit* and is 10 by default. The try's number is stored in the *Task.network_try_count*. If *network_try_count* reaches the *network_try_limit* the task is aborted.

When the task is abroted and there is method with name *task_<task-name>_fallback* then it is called and receives the failed task as first argument.

Also, it happens that you need to put task back to task queue even it was not failed due to the network error. For example, the response contains captcha challenge or other invalid data reasoned by the anti-scraping protection. You can control number of such tries. Max tries number is configured by *Spider.task_try_count*. The try's number is stored in *Task.task_try_count*. Keep in mind, that you have to increse *task_try_count* explicitly when you put task back to task queue.

```
def task_google(self, grab, task):
    if captcha_found(grab):
        yield Task('google', url=grab.config['url'],
                   task_try_count=task.task_try_count + 1)

def task_google_fallback(self, task):
    print 'Google is not happy with you IP address'
```

Manual Processing of Failed Tasks

You can disable default mechanism of processing failed tasks and process failures manually. Use *raw=True* parameter in Task contstructor. If the network request would fail then the grab object passed to the handler would contain information about failure in two attributes: *grab.response.error_code* and *grab.response.error_msg*

See example:

```
class TestSpider(Spider):
    def task_generator(self):
        yield Task('page', url='http://example.com/', raw=True)

    def task_page(self, grab, task):
        if grab.response.error_code:
            print('Request failed. Reason: %s' % grab.response.error_msg)
        else:
            print('Request completed. HTTP code: %d' % grab.response.code)
```

Error Statistics

After spider has completed the work or even in the process of working you can receive the information about number of completed requests, failed requests, number of specific network errors with method *Spider.render_stats*.

1.3 API Reference

Using the API Reference you can get an overview of what modules, classes, and methods exist, what they do, what they return, and what parameters they accept.

1.3.1 Module grab.base

Here is the heart of the library, the Grab class.

class `grab.base.`**Grab**(*document_body=None*, *transport='grab.transport.curl.CurlTransport'*, ***kwargs*)

__init__(*document_body=None*, *transport='grab.transport.curl.CurlTransport'*, ***kwargs*)
Create Grab instance

setup_transport(*transport_param*)

reset()
Reset all attributes which could be modified during previous request or which is not initialized yet if this is the new Grab instance.

This methods is automatically called before each network request.

clone(***kwargs*)
Create clone of Grab instance.

Cloned instance will have the same state: cookies, referrer, response document data

Parameters ****kwargs** – overrides settings of cloned grab instance

adopt(*g*)
Copy the state of another *Grab* instance.

Use case: create backup of current state to the cloned instance and then restore the state from it.

dump_config()
Make clone of current config.

load_config(*config*)
Configure grab instance with external config object.

setup(***kwargs*)
Setting up Grab instance configuration.

go(*url*, ***kwargs*)
Go to `url`

Args:

url could be absolute or relative. If relative then t will be

appended to the absolute URL of previous request.

download(*url*, *location*, ***kwargs*)
Fetch document located at `url` and save to to `location`.

request(***kwargs*)
Perform network request.

You can specify grab settings in `**kwargs`. Any keyword argument will be passed to `self.config`.

Returns: `Document` objects.

fake_response(**args*, ***kwargs*)

load_proxylist(**args*, ***kwargs*)

change_proxy()
Set random proxy from proxylist.

make_url_absolute(*url*, *resolve_base=False*)
Make url absolute using previous request url as base url.

clear_cookies()
Clear all remembered cookies.

load_cookies(**args*, ***kwargs*)

dump_cookies(**args*, ***kwargs*)

setup_with_proxyline(*line*, *proxy_type='http'*)

grab.base.**BaseGrab**
alias of *Grab*

1.3.2 Module grab.error

Custom exception classes for Grab.

class grab.error.**GrabError**
All custom Grab exception should be children of that class.

class grab.error.**GrabNetworkError**
Raises in case of network error.

class grab.error.**GrabTimeoutError**
Raises when configured time is outed for the request.

In curl transport it is CURLE_OPERATION_TIMEDOUT (28)

class grab.error.**DataNotFound**
Raised when it is not possible to find requested data.

class grab.error.**GrabMisuseError**
Indicates incorrect usage of grab API.

class grab.error.**GrabConnectionError**
Raised when it is not possible to establish network connection.

In curl transport it is CURLE_COULDNT_CONNECT (7)

class grab.error.**GrabAuthError**
Raised when remote server denies authentication credentials.

In curl transport it is CURLE_COULDNT_CONNECT (67)

class grab.error.**GrabTooManyRedirectsError**
Raised when Grab reached max. allowd number of redirects for one request.

class grab.error.**GrabDeprecationWarning**
Raised when some deprecated feature is used.

class grab.error.**GrabInvalidUrl**
Raised when Grab have no idea how to handle the URL or when some error occurred while normalizing URL e.g. IDN processing.

1.3.3 Module grab.cookie

This module contains some classes to work with cookies.

grab.cookie.**create_cookie**(*name*, *value*, *domain*, *httponly=None*, ***kwargs*)
Creates *cookielib.Cookie* instance

class grab.cookie.**CookieManager**(*cookiejar=None*)
Each Grab instance has *cookies* attribute that is instance of *CookieManager* class.

That class contains helpful methods to create, load, save cookies from/to different places.

__init__(*cookiejar=None*)

set(*name*, *value*, *domain*, ***kwargs*)
Add new cookie or replace existing cookie with same parameters.

Parameters

- **name** – name of cookie
- **value** – value of cookie
- **kwargs** – extra attributes of cookie

update(*cookies*)

clear()

__getitem__(*key*)
Implements dict interface, allows to get cookie value by its name.

items()

get_dict()

load_from_file(*path*)
Load cookies from the file.

Content of file should be a JSON-serialized list of dicts.

save_to_file(*path*)
Dump all cookies to file.

Cookies are dumped as JSON-serialized dict of keys and values.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

g

grab.base, 39
grab.cookie, 41
grab.error, 40

Symbols

__getitem__() (grab.cookie.CookieManager method), 41
__init__() (grab.base.Grab method), 39
__init__() (grab.cookie.CookieManager method), 41

A

adopt() (grab.base.Grab method), 39

B

BaseGrab (in module grab.base), 40

C

change_proxy() (grab.base.Grab method), 40
clear() (grab.cookie.CookieManager method), 41
clear_cookies() (grab.base.Grab method), 40
clone() (grab.base.Grab method), 39
CookieManager (class in grab.cookie), 41
create_cookie() (in module grab.cookie), 41

D

DataNotFound (class in grab.error), 40
download() (grab.base.Grab method), 39
dump_config() (grab.base.Grab method), 39
dump_cookies() (grab.base.Grab method), 40

F

fake_response() (grab.base.Grab method), 40

G

get_dict() (grab.cookie.CookieManager method), 41
go() (grab.base.Grab method), 39
Grab (class in grab.base), 39
grab.base (module), 39
grab.cookie (module), 41
grab.error (module), 40
GrabAuthError (class in grab.error), 40
GrabConnectionError (class in grab.error), 40
GrabDeprecationWarning (class in grab.error), 40
GrabError (class in grab.error), 40
GrabInvalidUrl (class in grab.error), 41
GrabMisuseError (class in grab.error), 40
GrabNetworkError (class in grab.error), 40
GrabTimeoutError (class in grab.error), 40
GrabTooManyRedirectsError (class in grab.error), 40

I

items() (grab.cookie.CookieManager method), 41

L

load_config() (grab.base.Grab method), 39
load_cookies() (grab.base.Grab method), 40
load_from_file() (grab.cookie.CookieManager method), 41
load_proxylist() (grab.base.Grab method), 40

M

make_url_absolute() (grab.base.Grab method), 40

R

request() (grab.base.Grab method), 39
reset() (grab.base.Grab method), 39

S

save_to_file() (grab.cookie.CookieManager method), 41
set() (grab.cookie.CookieManager method), 41
setup() (grab.base.Grab method), 39
setup_transport() (grab.base.Grab method), 39
setup_with_proxyline() (grab.base.Grab method), 40

U

update() (grab.cookie.CookieManager method), 41